Lx: A Technology Platform for Customizable VLIW Embedded Processing

Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher,
Giuseppe Desoli, Fred (Mark Owen) Homewood*

Hewlett-Packard Laboratories (Cambridge, MA)
*STMicroelectronics (Cambridge, MA)
{frb,gbrown,jfisher,desoli}@hpl.hp.com, fred@bristol.st.com

ABSTRACT

Lx is a scalable and customizable VLIW processor technology platform designed by Hewlett-Packard and STMicroelectronics that allows variations in instruction issue width, the number and capabilities of structures and the processor instruction set. For Lx we developed the architecture and software from the beginning to support both scalability (variable numbers of identical processing resources) and customizability (special purpose resources).

In this paper we consider the following issues. When is customization or scaling beneficial? How can one determine the right degree of customization or scaling for a particular application domain? What architectural compromises were made in the Lx project to contain the complexity inherent in a customizable and scalable processor family?

The experiments described in the paper show that specialization for an application domain is effective, yielding large gains in price/performance ratio. We also show how scaling machine resources scales performance, although not uniformly across all applications. Finally we show that customization on an application-by-application basis is today still very dangerous and much remains to be done for it to become a viable solution.

1. INTRODUCTION

Dataquest estimates that the embedded processor market should grow from $7.5 billion in 1998 to $26 billion by 2002. This market space is seeing an increasing number of competitors ranging from companies implementing variations of traditional embedded processor architectures (such as ARM and MIPS), to more aggressive startups introducing their own new ISA (such as ARC Cores and Tensilica).

At the same time, the complexity of embedded applications is escalating considerably, and it is not uncommon to find many hundred of thousands of lines of high-level language code in embedded products such as printers or mobile phones. Time-to-market is also becoming a primary concern, as the lifetime of embedded products constantly shrinks to keep pace with evolving standards, new user needs and performance requirements.

ISCA 00 Vancouver, British Columbia Canada

The combination of application complexity and time-to-market considerations is what makes a software-based approach to embedded systems particularly appealing today. Ideally, embedded system designers would like to have a single processing platform where high performance digital signal processing capability (for real-time signal processing), is coupled to microprocessor functionality (for general purpose processing tasks). This trend is what is causing the traditionally separated DSP and micro-controller domains to converge in an increasingly large number of products that are starting to be commercially offered.

Our approach is based on two concepts:

- A new clustered VLIW core architecture and microarchitecture specialized to an application domain that ensures scalability and customizability
- A toolchain based on aggressive ILP compiler technology that gives the user a uniform view of the platform at the programming language level.

The technology we are developing is called "*Lx*", we are doing it in a production environment, most pieces have already been developed, and products are expected in the near future.

The reasons for developing a new ISA come from the observation that existing architectures are not scalable in width and customization areas are limited. Existing ISAs are either too specialized (most DSP processors) or too general (general-purpose platforms like ARM and MIPS).

We believe that the combination of: clustering, VLIW with precise interrupts, a slim and scalable microarchitecture, and interesting memory hierarchies constitute a novel technology platform.

1.1 Convergence of Embedded Technologies

DSP and micro-controllers are converging in the high-end markets. This new batch of processors include a combination of features from the DSP domain, such as low-overhead looping, rich set of addressing modes, special purpose arithmetic operations and formats, etc. At the same time, they usually include a more RISC-like set of instructions (sometimes in a different mode), to ease high-level (C or C++) code development, to support system code and multitasking OS's and in general to be able to implement much larger applications in the same platform. In the following we discuss the subset of announced DSPs and configurable RISC cores that have the most commonality with the Lx architecture.

Over the past several years a number of semiconductor manufacturers have announced high performance embedded VLIW cores

and processors. These include the Motorola/Lucent *StarCore*, the TI *C6xxx* family, and the Philips *Trimedia*. Of these, all but the *StarCore* are currently in production; however, only the TI C6 family has apparently shipped in large volumes. In addition, STMicroelectronics has announced the ST100 DSP, which has a "VLIW mode" for key inner loops.

- The announced *StarCore* architecture [13] is a "natural" VLIW extension of traditional DSPs – the basic operations supported are optimized for DSP applications with the ability to issue multiple operations simultaneously. While not as register starved as previous DSPs, the available 16 data registers are likely to make the compiler's task difficult.
- The *TI C6* family [15] is significantly closer than the StarCore to a general-purpose processor. C6 presents some difficulties for real time applications because, for example, software pipelining using modulo scheduling is evidently not interruptible and interruptible code requires hazard-free register usage. This may cause significant register pressure for the compiler. In contrast, the Lx was designed to be interruptible, and all code generated by the compiler is hazard free.
- The *Phillips Trimedia* processors [12] are the most ambitious of the currently available embedded VLIW processors. Its instruction set is quite rich, includes floating point and multimedia instructions and full predication. In contrast, Lx has a modest set of basic instructions that allow future family members to be customized for specific domains.

In addition to embedded VLIW cores, several configurable processors were recently announced. The most visible of these are the *Tensilica Xtensa* architecture and *ARC Cores*. The *Tensilica* processor [14] takes an *a-la-carte* approach with some support for custom instructions. The designer has the ability to choose from optional functional units, memory interfaces, and peripherals. In addition, the toolchain supports user-defined instructions that are defined using a simple RTL. The basic processor is a simple single issue RISC core. The support for user defined instructions is consistent with that provided by the Lx toolchain. Thus the *Tensilica* processor is customizable, but not scalable. The *ARC* core [8] is a much simpler processor than the *Xtensa* providing a modest set of synthesis time options to allow some choice among a set of predefined instructions and peripherals.

1.2 Competing Technologies

It is important to compare customizable VLIW architectures to other competing high-performance computing technologies in the embedded space. Table 1 summarizes the situation and shows how the advantages of *high performance, ease of use* and *flexibility* uniquely position this technology. This is particularly true in a world where time-to-market is rapidly becoming the dominant factor in the success of a new technology.

2. SCALABILITY AND CUSTOMIZABILITY

For an embedded architecture, we define scalability as the ability to vary the number of existing resources, and customizability as the ability to add new resources. In the Lx family, scalability includes varying the instruction issue width and the mix of operations that may be issued simultaneously. Scaling in this sense does not change the set of operations in the ISA; however, for statically scheduled architectures, scaling implies changing the set of legal programs.

The Lx platform was developed in the belief that large performance advantages are available if we can change both the quantity of computation available by scaling, and the actual computations done efficiently, by customizing.

In the rest of the paper we will show that, although it is technically possible to customize on an application-by-application basis, today it makes more sense to customize on an application-area (or *domain*) basis. For example, we can picture one architecture family customized for digital consumer (with implementation scaled within that architecture family), another for printing applications, and so on.

It is hard to quantify the advantages of scaling and customizing, as other works [4] have shown. Sometimes, a large factor speedup can be obtained via a very special, bit-twiddling operation (MMX-style extension fall into this category). Sometimes doubling the functional units doubles performance, sometimes it adds no performance at all. Sometimes customization can have dramatic effects on inner loops, but little effect on the whole application (Amdahl's Law). The remainder of the paper addresses some of these issues and highlights the customization directions that we believe are more promising in the high-performance embedded domain.

3. THE Lx CORE ARCHITECTURE

Lx is a family of embedded cores designed by Hewlett-Packard Laboratories and STMicroelectronics. It is a scalable platform where developers can pick the family member based on cost/performance considerations for their application. For the first generation of the family, scalability is planned to span from 1 to 4 clusters (i.e., 4 to 16 issued instructions per cycle). In addition to simple scalability, Lx was designed to be customizable to specific applications or application areas through the addition of application-specific operations.

Speed

Technology	Performance attainable	Time until running	Time to high performance	Time to change code functionality
ASIC	Very High	Very Long	Very Long	Impossible: redesign
DSP / ASIP	High	Long	Long	Long
Custom VLIW	High	Short	Short	Short
RISC	Low-Medium	Very Short	Not Attainable	Very Short

Flexibility

Table 1 High Performance Computing Technologies for Embedded Systems

Lx is a statically scheduled VLIW architecture, thus providing the most computation at a given silicon area. The VLIW approach also yields the system advantages of a RISC instruction set, such as fast interrupts, normal debugging, and so on. Despite a VLIW instruction set, the operation encoding enables Lx code size to be competitive with other 32-bit embedded platforms.

Lx comes with a commercial software toolchain, where no visible changes are exposed to the programmer when the core is scaled and customized. The toolchain includes sophisticated ILP compiler technology (derived from the Multiflow compiler [7]) coupled with widely accepted GNU tools and libraries. The Multiflow compiler includes most traditional high-level optimizations algorithms and aggressive code motion technology based on Trace Scheduling [5]. It is considered one of the most optimized ILP compilers commercially available and is still used broadly in the computer industry.

3.1 Multi-cluster Organization

Lx is a *Multi-cluster architecture* [3], as shown in Figure 1. Lx clusters are composed of a mix of *Register Banks*, *Constant Generators* (immediate operands) and *Functional Units*. Different clusters may have different unit/register mixes, but a single PC and a unified I-cache control them all, so that they run in lockstep. Likewise, the same execution pipeline drives all clusters. Inter-cluster communication, achieved by explicit *register-to-register move*, is compiler-controlled and invisible to the programmer.

At the multi-cluster level, the architecture specifies:

- The instruction delivery mechanism, to get instructions from the cache to the clusters' data-path. The assumption is that all clusters feed synchronously from the same logical instruction cache, which will be implemented differently depending on technology and cost considerations. Likewise, the pipeline may incur additional decoding cycles beyond certain clustering limits.
- The inter-cluster communication mechanism, to transfer data among clusters. Lx defines a scalable and flexible communication mechanism based on a simple pair of *send-receive* instruction primitives that move values among registers. The send-receive method is scalable, as it does not depend on the number of clusters; and is flexible, as it leaves room for multiple microarchitecture implementations (bus-based, with private cluster wires, etc.).
- The data-cache organization, to establish main memory coherency in the presence of multiple memory accesses. We investigated two models: a MESI-like synchronization mechanism for multiple independent caches, and a pseudo-multi-ported cache implemented with multiple interleaved banks. A discussion of the tradeoffs of these and other mechanisms is beyond the scope of the paper.

3.2 The Organization of a Single-cluster

An Lx cluster (Figure 2) is a 4-issue VLIW core composed of four 32-bit integer ALUs, two 16x32 multipliers, one Load/Store Unit and one Branch Unit. The cluster also includes 64 32-bit General-purpose registers and 8 1-bit branch registers (used to store branch condition, predicates and carries). Instructions allow two long immediates per cycle.

The ISA is a very simple integer RISC instruction set with minimal "predication" support through *select* instructions. The memory repertoire includes *base+offset* addressing, allows speculative execution (*dismissible loads,* handled by the protection unit) and software *prefetching*.

Lx includes a two-level code compression scheme. The instruction cache is compressed so that unused operation slots do not consume space in the instruction encoding. In addition, we developed an aggressive compression scheme where binaries are compressed with a Huffman-like technique, and blocks of instructions are decompressed on I-cache refill (discussed in section 3.4).

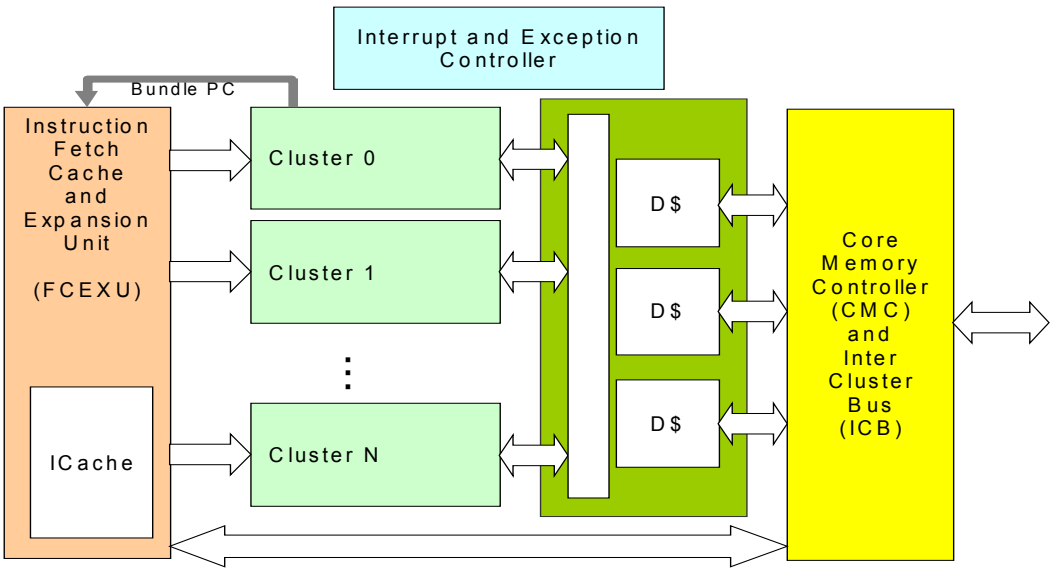


Figure 1 The structure of a multi-cluster Lx architecture

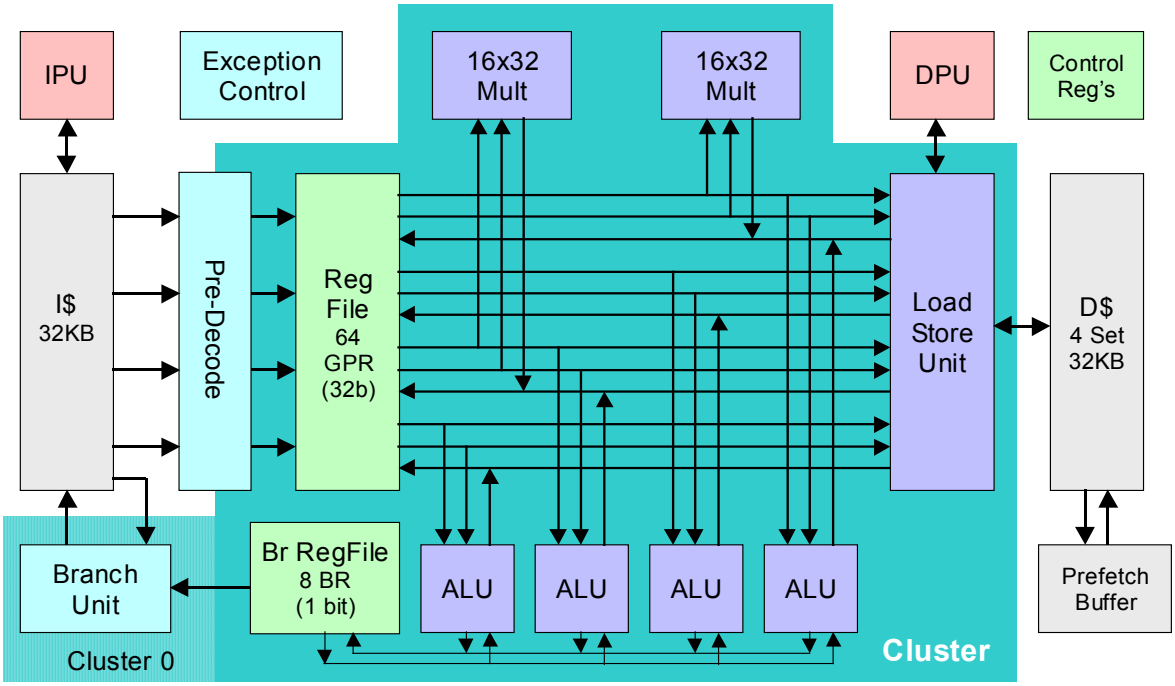


Figure 2 Structure of a single-cluster Lx. The shaded area in the middle contains the "proper" cluster resources. The other components (Caches, Instruction/Data Protection Units, Control Registers, Exception Control) are shared in a multi-cluster configuration. Branch Units are restricted only to cluster 0.

The control unit supports a "two-step" branch architecture, where compare and branch operations are de-coupled and the compare-branch latency is exposed to the compiler. The ISA includes a complete set of compare and logic operations and a separate set of 8 1-bit branch registers for conditions that allow us to prepare multiple branches (up to eight). There are no architecturally visible delay slots after a taken branch so that - for example - dynamic branch prediction could be added (if needed) in follow-up microarchitecture implementations.

3.3 Pipeline, Memory, Interrupts

Lx has a classical six-stage pipeline: **`F D R E1 E2 W`**. It is a simple in-order pipeline: the exception point is at E2 and all commit points are delayed until after E2 so that all units commit their results to the register file in order. This allows us to have a very clean exception model, despite the complexity of a wide-issue machine. The data-path is fully bypassed from E1 and E2 and completely hidden at the architecture level (i.e. the results of single cycle operations are available to operations in the following pipeline stage).

The data cache is a 32KB, 4-way associative, write-back array with load/store allocation. It includes an 8-entry software controlled *Prefetch Buffer* that acts as a small level-2 fully associative cache where requests currently in cache or in prefetch buffer are dropped and data is copied from the buffer to the cache during a subsequent miss.

The memory controller includes a simple *Protection Unit* that supports segment-based protection regions, speculative loads (where traps are dismissed) and is easily extendable to a full MMU for customers that require it.

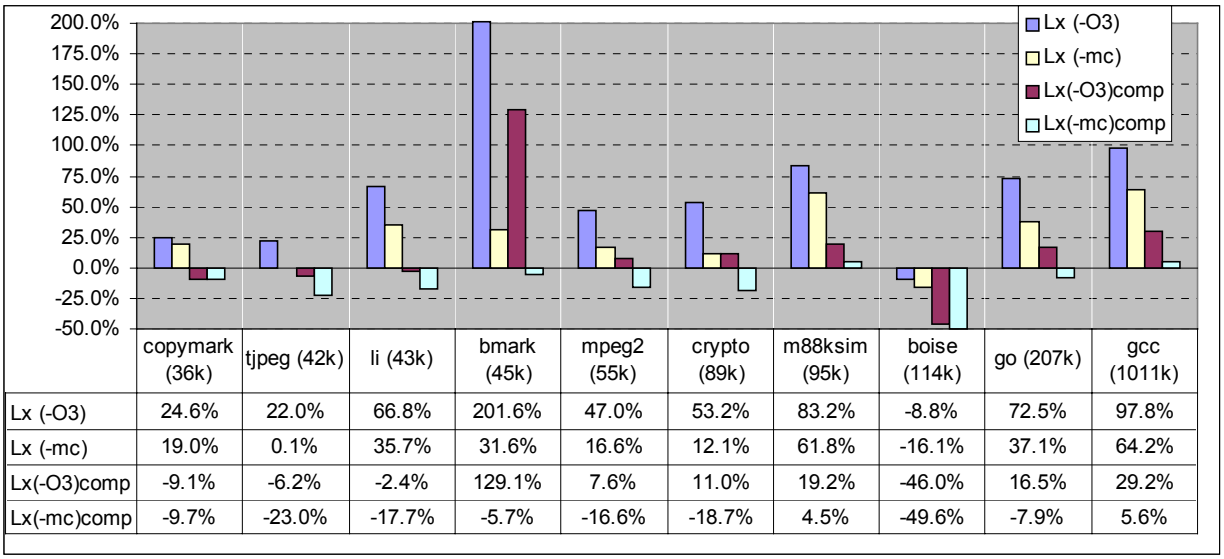
The memory model is unified, including internal, external memory, peripherals and control registers (that are mapped into the upper 4K page). The core memory communicates with external memory and the peripheral controller using a VSI-like system-on-chip interface.

The interrupt controller supports the minimal set of required exceptions: illegal instruction, access violation and misaligned access. For all of these, software recovery is supported ("precise" model). Breakpoints are implemented with hardware support. In addition to exceptions, the first Lx core supports one hardware interrupt source: multiple priorities and interrupts sources are considered part of the customization layer. In this way, we can achieve a rather fast exception/interrupt response time, which is approximately 6 cycles to get to a cached exception handler.

3.4 Code Density

Many critics of VLIW technology cite code density as a primary disadvantage, which is indeed true for naïve VLIW implementations. However, if we try to break down the causes of potential increases in code size, we can see that they fall into three main categories:

1. Sparse ILP encoding. A naïve VLIW implementation would keep a one-to-one correspondence between functional units and instruction slots (also called *syllables*). This introduces "horizontal" no-ops for unused units, and is probably the largest source of inefficiency: it impacts both the instruction cache and main memory. However, all VLIW implementations—from the early Multiflow Trace [1] and Cydrome [9]



	copymark (36k)	tjpeg (42k)	li (43k)	bmark (45k)	mpeg2 (55k)	crypto (89k)	m88ksim (95k)	boise (114k)	go (207k)	gcc (1011k)
Lx (-O3)	24.6%	22.0%	66.8%	201.6%	47.0%	53.2%	83.2%	-8.8%	72.5%	97.8%
Lx (-mc)	19.0%	0.1%	35.7%	31.6%	16.6%	12.1%	61.8%	-16.1%	37.1%	64.2%
Lx(-O3)comp	-9.1%	-6.2%	-2.4%	129.1%	7.6%	11.0%	19.2%	-46.0%	16.5%	29.2%
Lx(-mc)comp	-9.7%	-23.0%	-17.7%	-5.7%	-16.6%	-18.7%	4.5%	-49.6%	-7.9%	5.6%

Figure 3 Lx code growth compared to StrongARM (SA-110), expressed as % code size increase (decrease when negative) vs. the SA-110. Lx numbers include, uncompressed code compiled for performance (*-O3*), minimum size (*-mc*) and then the corresponding compressed sizes.

to the latest TI C6xxx [15] and IA-64 [11]—provide some way to remove this inefficiency by means of a careful encoding that avoids explicit no-ops when units are unused. Techniques range from using template bits that encode the syllables in a bundle (Multiflow, IA-64), to run-length coding of no-ops (Cydrome), and so on. In Lx we achieve the same results simply by using an "end-of-bundle" bit.

2. RISC encoding and exposed latencies. The encoding of a general-purpose 32-bit RISC processor is intrinsically sparser than traditional CISC or DSP processors. This is amplified in non-scoreboarded VLIW architectures where latencies are exposed at the ISA level. However, several techniques exist to mitigate this phenomenon. Some embedded architectures adopt simplified forms of the instruction set in size-critical areas (for example: MIPS-16 and ARM-Thumb), others use a more systematic compression system to decompress lines on instruction cache misses (for example: the *CodePack* system for IBM PowerPC [6]). For Lx, we adopt a philosophy similar to that of the IBM *CodePack* system, where the code is compressed by software (with a Huffman-like algorithm after linking) and decompressed on-demand on an instruction cache miss by a simple hardware block connected to the external memory bus.

3. Compiler-driven code expansion. This is by far the hardest factor to quantify. Many techniques that expose ILP tend to grow code size, regardless of the architectural style. These include: loop unrolling; region-scheduling compensation code and global code motion; procedure inlining, cloning and specialization; and so on. However, many of these techniques need only be applied aggressively to the computational kernels of the application and—when needed—the user can guide the compiler heuristics for space/time optimizations.

Figure 3 shows code density figures for Lx in comparison to another 32-bit embedded platform (a StrongARM SA-110) at comparable levels of compiler optimizations (benchmarks are described in the following sections). As we can see, Lx code size is very competitive even in the presence of high levels of optimization. If we exclude *bmark,* where the code is heavily unrolled and the expansion is slightly above three, the rest of the benchmarks score between 25% and 100% code increase, with an average of 48%. If we apply compression, the overhead goes down significantly, to an average of 14.9%. If we compile for minimal code size (still at a reasonable optimization level), the average increase is 26%. Somewhat surprisingly, this turns into a code size decrease of -14% when we apply cache-line compression.

The memory savings for the code compression algorithm averages 32% of code reduction for optimized code, indicating that we can exploit a fair amount of redundancy in the RISC/VLIW encoding. These values are consistent with IBM CodePack results and academic studies 0, and we believe that this is probably the most effective and least invasive way to attack the code size problem, when necessary.

Note that compression would benefit StrongARM as well. Such a capability is not currently offered, since StrongARM code size is considered respectable for embedded applications. This means that VLIW code size can be brought down to the level of more traditional RISC processors, perceived to be adequate in this respect.

4. SPECIALIZATION AND SCALABILITY

To measure Lx performance on its target application domain, we collected a set of representative programs that include audio manipulation, printing pipelines, color processing, cryptography, video and still image compression and decompression. The domain benchmarks were optimized at the C source level (no assembler) by adding compiler pragmas (unrolling and aliasing directives) and in some cases restructuring the loops to expose

Name	Description	Name	Description
bmark	Printing imaging pipeline (optimized)	*boise*	Printing rendering pipeline (C++)
copymark	Color copier pipeline (optimized)	*dhry*	Dhrystone 1.1 and 2.1 benchmark
crypto	Cryptography code (optimized)	*gcc*	SPECINT'95 GNU cc compiler
csc	Color-space conversion (optimized)	*go*	SPECINT'95 game of GO
mpeg2	MPEG-2 decoder (optimized)	*li*	SPECINT'95 LISP interpreter
tjpeg	JPEG-like coder/decoder (optimized)	*m88ksim*	SPECINT'95 M88000 simulator
adpcm	ADPCM audio coder/decoder	*gs*	Ghostscript PostScript interpreter
Application Domain		***Reference Benchmarks***	

Table 2 The benchmark set.

more ILP. Our source-level optimizations improved performance also for our reference platforms, although the reference compiler (gcc) did not take advantage of the pragmas.

To evaluate how Lx behaves on programs outside the target domain, we also added a set of unmodified reference benchmarks from the SPECINT'95 suite (*gcc, go, li, m88ksim*[1]) and some other well-known public code (*ghostscript* and *dhrystone*), as well as a rendering program in C++ not optimized at the source level. Table 2 describes the programs in our benchmark set.

To evaluate the benefits of the domain-specific specializations in Lx, we show performance measurements relative to a baseline configuration. Our baseline numbers are for a Pentium-II at 333 MHz (measured on an HP Kayak XU PC workstation), compiled with *gcc* (v. 2.95, all optimizations), using *cygwin* libraries. Use of MMX instructions was ruled out of these experiments because they are non-portable, and their use is much more labor-intensive than the code changes we permitted for Lx.

In all graphs, we show measurements for a more typical high-performance 32-bit embedded processor, the StrongArm SA-110 at 275MHz, measured on a Corel NetWinder machine, compiled with *gcc* (version 2.8.1, all optimizations), using *linux* libraries.

Lx performance is measured on a cycle-accurate simulator (validated against a Verilog model) that includes cache, bus and external memory measurements. The Lx C compiler is a descendant of the Multiflow compiler and uses the GNU *newlib* libraries (the C++ benchmark was translated by *cfront*). The Lx memory system is a typical embedded system configuration: a 100MHz unified code/data memory bus, with 6-1-1-1[-3] DRAM bursts.

As we can see from the following graphs, Lx is extremely fast on *compute-intensive, loop-dominated* or *hand-tuned code*, such as MPEG decoding, JPEG encoding/decoding, DSP algorithms, cryptography, and so on. On these applications, Lx is more than 2x faster than a PII-333 at a tiny fraction of the area, and factors (4x-8x) faster than a SA100-275 at a comparable area. On the other hand, Lx performance is "average" on *control-dominated, non-optimized code,* or *code with a strong unpredictable component*, such as interpreters (*gs*), compilers (*gcc*), simulators (*m88ksim*), rule-based "AI" (*go*), C++ (*boise*), and so on.

In the following two sections, we are interested in evaluating how performance scales when we apply variations along two different directions: clock frequency and issue width.

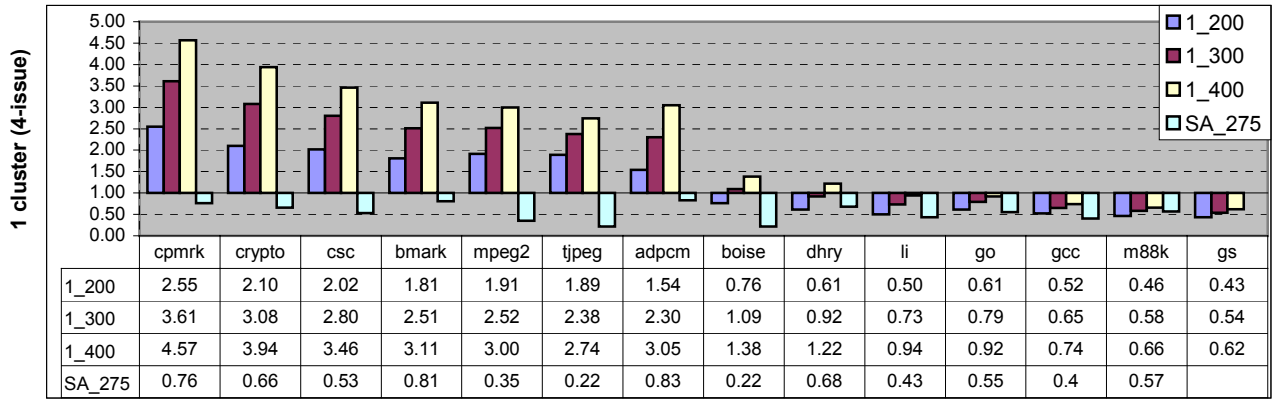
4.1 Scaling Clock Frequency

As a rule of thumb, power consumption in a microprocessor grows linearly with frequency and quadratically with voltage. Usually, lowering frequency allows operating at smaller voltages, and this has a cubic effect on power savings. Therefore, in embedded domains with a limited energy budget, scaling clock frequency may not always be the preferred solution and it is important to evaluate the performance benefits. Figure 4 shows performance numbers for Lx at 3 different clock frequencies (200, 300 and 400 MHz) for the benchmark set. We can see that for the applications in the target domain, performance scales almost linearly with clock frequency. This remains true also for wider issues machines (we show a 2-cluster 8-issue and a 4-cluster 16-issue Lx). Note that the external memory hierarchy was fixed for all the experiments, so that the overall system cost is only marginally affected by the change in clock speed.

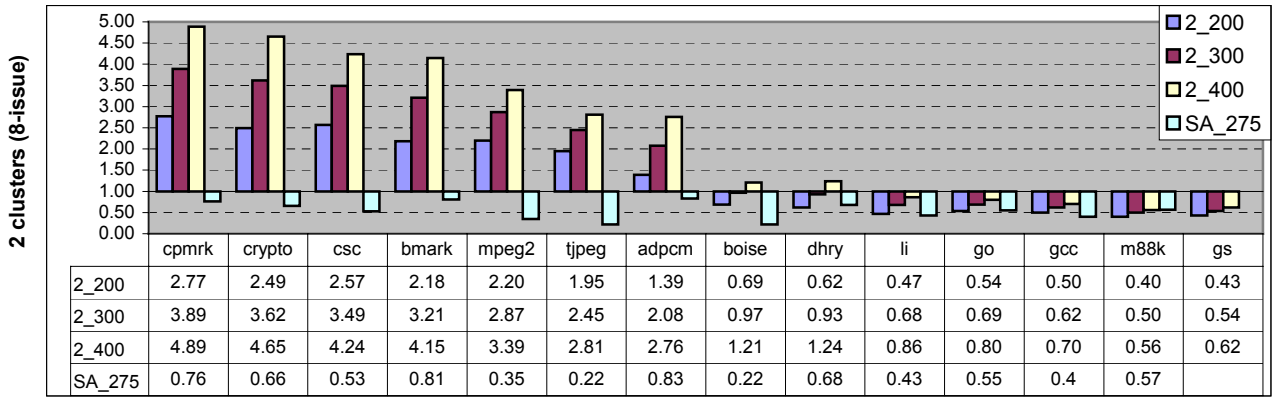
If we consider the collection of general-purpose applications, we see that there is not much we can do by increasing processor speed. We can observe that Lx performance is competitive and often better than an embedded processor at a similar cost range. Finally, Lx still lies about 30% below a workstation-class processor (like the Pentium-II) that can afford to adopt more expensive features like larger caches, multiple level memory hierarchies and aggressive dynamic branch prediction.

For our experiments we chose a frequency range that is realistic for typical 0.25µ Lx implementations, between 200 and 400 MHz. At the same time, we kept the external memory interface constant assuming a 100MHz unified external memory bus. This means that, for example, a data cache miss (32 bytes/line) takes 25 cycles on a 200 MHz Lx, 36 cycles on a 300 MHz Lx and 47 cycles on a 400 MHz Lx. Similar considerations apply for the instruction cache misses. Level-2 caches are rarely used in embedded systems due to their impact on overall system cost and we chose not to model them.

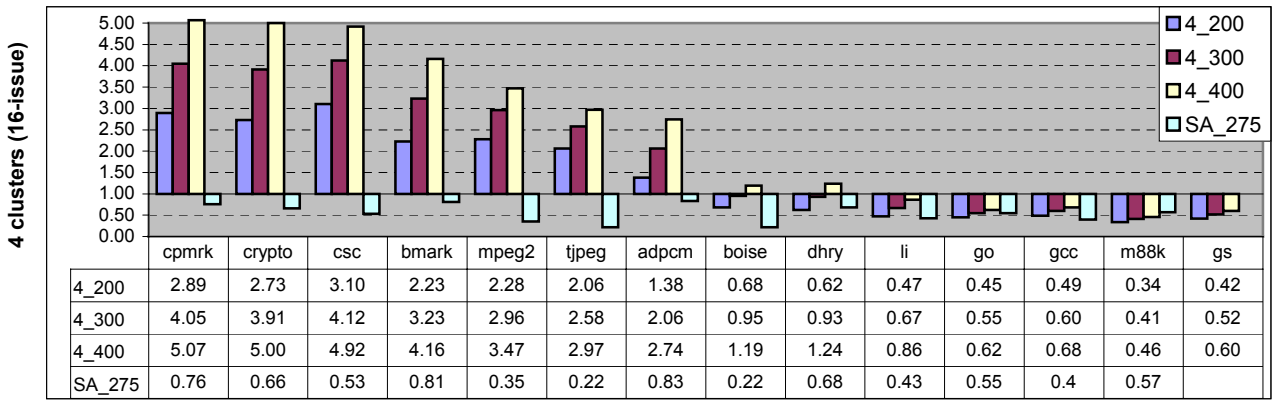
[1] From the SPECINT'95 suite we left out *perl* and *vortex* since they are not relevant to the embedded domain; *compress* and *ijpeg* since the compression decompression domain is already well-represented by the other benchmarks.



	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
1_200	2.55	2.10	2.02	1.81	1.91	1.89	1.54	0.76	0.61	0.50	0.61	0.52	0.46	0.43
1_300	3.61	3.08	2.80	2.51	2.52	2.38	2.30	1.09	0.92	0.73	0.79	0.65	0.58	0.54
1_400	4.57	3.94	3.46	3.11	3.00	2.74	3.05	1.38	1.22	0.94	0.92	0.74	0.66	0.62
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	



	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
2_200	2.77	2.49	2.57	2.18	2.20	1.95	1.39	0.69	0.62	0.47	0.54	0.50	0.40	0.43
2_300	3.89	3.62	3.49	3.21	2.87	2.45	2.08	0.97	0.93	0.68	0.69	0.62	0.50	0.54
2_400	4.89	4.65	4.24	4.15	3.39	2.81	2.76	1.21	1.24	0.86	0.80	0.70	0.56	0.62
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	



	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
4_200	2.89	2.73	3.10	2.23	2.28	2.06	1.38	0.68	0.62	0.47	0.45	0.49	0.34	0.42
4_300	4.05	3.91	4.12	3.23	2.96	2.58	2.06	0.95	0.93	0.67	0.55	0.60	0.41	0.52
4_400	5.07	5.00	4.92	4.16	3.47	2.97	2.74	1.19	1.24	0.86	0.62	0.68	0.46	0.60
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	

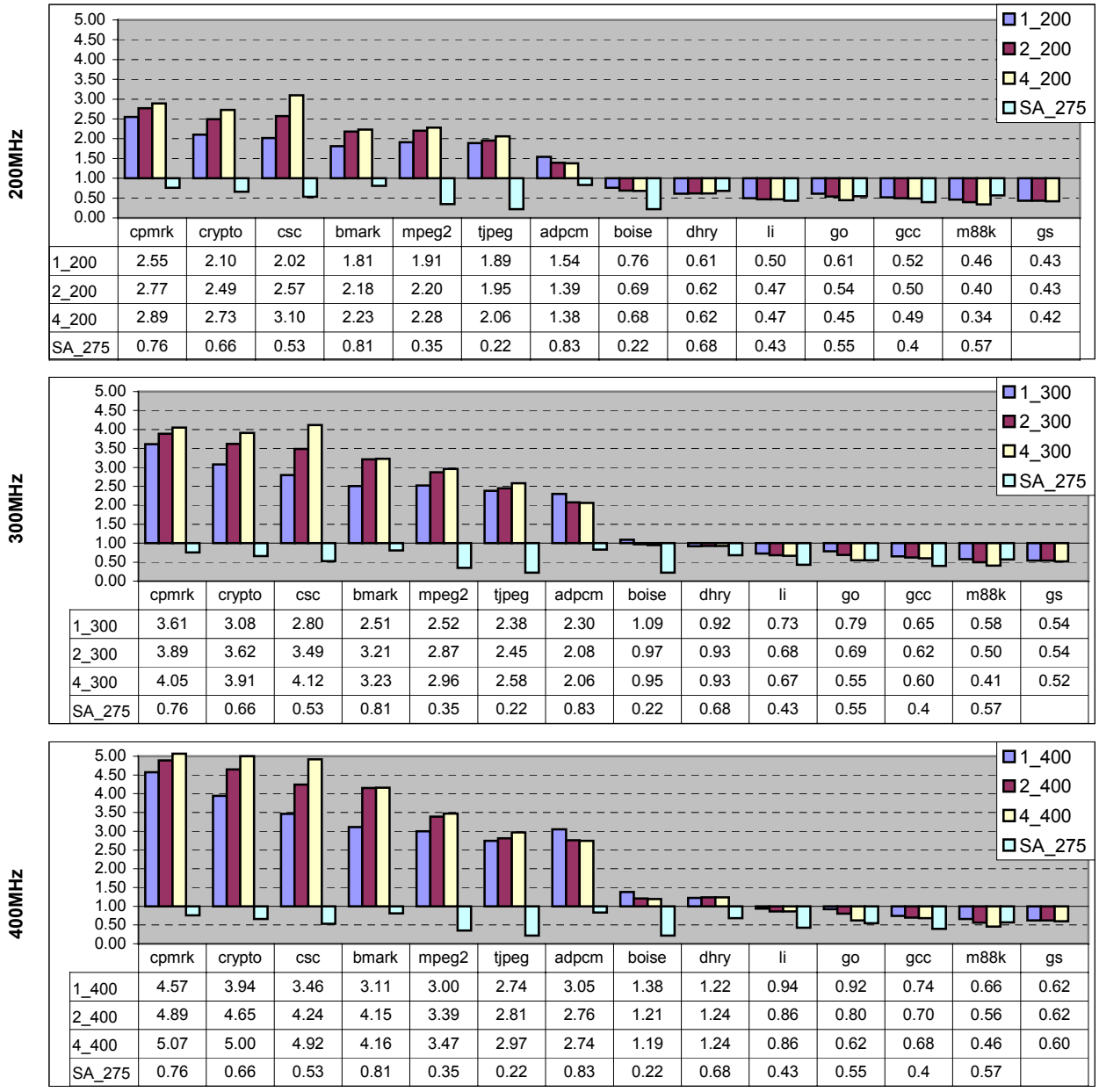
Figure 4 Lx performance chart: scaling clock frequency (200-300-400 MHz).
Performance is compared to a Pentium-II at 333 MHz, which is 1.00 on the vertical axis.
The SA_275 bars represent a 275MHz StrongARM.

4.2 Scaling Issue Width

In Figure 5 we present the same data of Figure 4 grouped by issue width. While increasing frequency and voltage has a cubic effect on power, power grows at most linearly with area increases. Functional units and registers represent a relatively small fraction of the overall processor area, so the effect on power consumption is marginal. Changing the issue width mostly affects processor cost, since the size of the data-path grows linearly with the number of clusters, and we assume that the bandwidth to the data cache also increases with the number of clusters. For these experiments we consider a pseudo-multi-ported data cache implemented through multiple interleaved banks, with a stalling mechanisms that can resolve bank conflicts in one cycle.

Here, we can see how scaling the issue width provides some advantages, but much less uniformly across the domain. In the target domain, doubling from 4-issue to 8-issue gives no more than a 25% improvement, and sometimes as little as 5-10%. For general-purpose code, wider issue above 4 is ineffective and sometimes detrimental (due to inter-cluster communication overhead, increased code size and data cache conflict stalls).

A detailed analysis of ILP saturation in this experiment is beyond



200MHz	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
1_200	2.55	2.10	2.02	1.81	1.91	1.89	1.54	0.76	0.61	0.50	0.61	0.52	0.46	0.43
2_200	2.77	2.49	2.57	2.18	2.20	1.95	1.39	0.69	0.62	0.47	0.54	0.50	0.40	0.43
4_200	2.89	2.73	3.10	2.23	2.28	2.06	1.38	0.68	0.62	0.47	0.45	0.49	0.34	0.42
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	

300MHz	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
1_300	3.61	3.08	2.80	2.51	2.52	2.38	2.30	1.09	0.92	0.73	0.79	0.65	0.58	0.54
2_300	3.89	3.62	3.49	3.21	2.87	2.45	2.08	0.97	0.93	0.68	0.69	0.62	0.50	0.54
4_300	4.05	3.91	4.12	3.23	2.96	2.58	2.06	0.95	0.93	0.67	0.55	0.60	0.41	0.52
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	

400MHz	cpmrk	crypto	csc	bmark	mpeg2	tjpeg	adpcm	boise	dhry	li	go	gcc	m88k	gs
1_400	4.57	3.94	3.46	3.11	3.00	2.74	3.05	1.38	1.22	0.94	0.92	0.74	0.66	0.62
2_400	4.89	4.65	4.24	4.15	3.39	2.81	2.76	1.21	1.24	0.86	0.80	0.70	0.56	0.62
4_400	5.07	5.00	4.92	4.16	3.47	2.97	2.74	1.19	1.24	0.86	0.62	0.68	0.46	0.60
SA_275	0.76	0.66	0.53	0.81	0.35	0.22	0.83	0.22	0.68	0.43	0.55	0.4	0.57	

Figure 5 Lx performance chart: scaling cluster width 1 to 4 (issues 4-wide, 8-wide and 16-wide)

the scope of this paper and the particular values clearly depend on the compiler, the coding style and the specific applications. However, what emerges is that some amount of restructuring is certainly necessary for applications to benefit from aggressive ILP. If we consider the benchmarks that perform best (*crypto, csc, bmark*, *mpeg2*), they were all optimized for ILP through compiler annotations and algorithm modifications

4.3 Putting it all together

From our analysis, we can draw the following conclusions:

- Specializing for an application domain pays off. In the Lx case (integer imaging and media-manipulation algorithms) we can get 4x-8x performance gains starting from C-level code with respect to a more general-purpose architecture at similar cost and technology.
- Scaling speed vs. power pays off fairly uniformly across the application domain and gains are almost linear in the considered frequency range.
- Scaling issue width vs. cost sometimes pays off, but yield smaller gains and not uniformly across all applications.
- Outside the application domain that we specialize for, we can still get performance that is comparable with a general-purpose architecture. This is very important in the real world, since it de-risks the introduction of a new technology by not

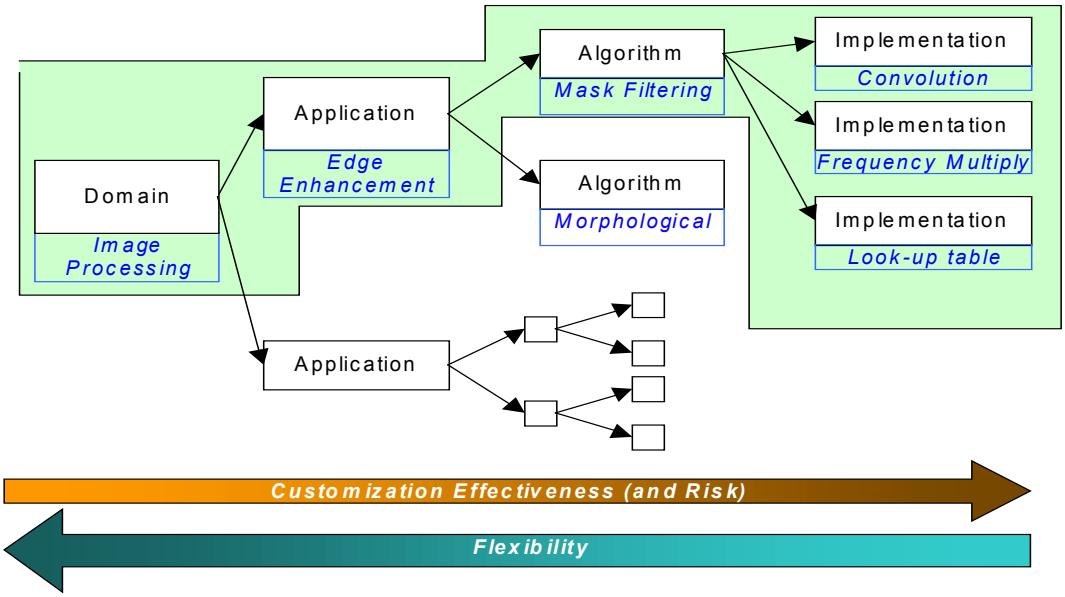


Figure 6 Example of different customization levels in the image-processing domain. One of the risks of over-customization comes from the difficulty of distinguishing between algorithms, implementations and implementation alternatives. This still requires the human intervention of an application expert.

tying its performance too closely to a narrow set of applications and attacking the Amdhal's Law problem. However, performance on general-purpose applications scales poorly with respect to issue width and clock speed, which is not surprising assuming a fixed memory hierarchy beyond the processor.

5. APPLICATION-SPECIFIC CUSTOMIZATION

If we look at the space between application *domains* and *implementations*, it is possible to conceive of customization at any level in the hierarchy (Figure 6). From more to less general levels of commitment, we can distinguish:

- *Domain-specific customization*. This is where a platform like Lx achieves the largest benefits, as we have seen in the previous sections. At the domain level, we can influence choices like the core ISA, the pipeline organization, the memory hierarchy, and the mix of resources. For example, the choice between an integer and floating-point data-path sits at this level. These choices impact the target market of the technology platform (DSP's, micro-controllers, very-low power, etc.)
- *Application-specific customization.* Within a domain, we can distinguish specific applications that are tied to individual products. Here it is conceivable to think about sizing and scaling the basic resources according to the overall characteristics of the application. For example, if our target product is a digital camera, we will have to stress low-power (at the expense of clock cycle speed), but at the same time we know that the application will likely contain a lot of potential ILP.
- *Algorithm-specific customization*. Even when the application is fixed, there is usually a wide choice of algorithms that yield similar results. The choice of the algorithm is usually driven by output quality, desired performance, implementation costs and other less quantifiable factors (like intellectual property issues). At the algorithm level, we can think of adding specific customization in the form of special computation instructions, storage organization (special memories) or other ad-hoc structures.
- *Implementation-specific customization*. For a given algorithm within an application, it is fairly common to have several software implementations that produce the same bit-by-bit result. For example, a simple 2D image filter can be implemented as a 2D convolution in the space domain or as a multiplication in the frequency domain; this is where specialized customization has larger benefits and where we can think of automating the process (as described for example in [4]). Unfortunately, this is also where such customization is riskiest, as we show in the following sections.

The two major trends that dominate the customization space are effectiveness and flexibility.

- Effectiveness. Obviously, the more we freeze our application/algorithm/implementation space, the higher the advantages of customization will be. At the same time, this goes against *time-to-market.* Probably the primary reason why developers prefer a software approach to a hardware design is the fact that with software they are free to change the application at the very last minute.

  A pure software approach also enables concurrent engineering between hardware and firmware design. By adding customization, the hardware (i.e., processor) design time gets in the critical path of the software design, that has to be completed by the time customization starts. Note that we can di-

minish this risk by reducing the design (and verification) time of a new custom processor, which is where automation helps. Finally, it is worth mentioning that rarely can the specialization for one specific implementation be leveraged into another one, and this also increases the risk of customization.

- Flexibility. In order to maintain flexibility we can be more general in our customization, at the expense of peak performance for some specific algorithm and implementation. At the extreme, we can design a completely general-purpose processor. We also note that the more we raise the generality of our customizations, the harder it is to design a process that can do that automatically, although we have a fair chance that any reasonable algorithm within the domain will benefit from the added features. A carefully designed generic MMX-style (micro-SIMD) instruction set extension is a good example of this.

5.1 A Customization Case Study: MD5 Encryption

To illustrate the concepts expressed in the previous section, we use a common encryption algorithm called MD5 (*Message Digest*). MD5 is a one-way hash function that produces a 128-bit hash of an input message and is commonly used in secure transactions and in the generation of signatures [10]. MD5 is interesting in our context since it represents a computationally intensive real application that is at the same time a good candidate for heavy customization.

The basic computation of MD5 is based on four elementary operators (we use C notation):

```
F(x,y,z) = (x & y) | (~x & z)
G(x,y,z) = (x & z) | (y & ~z)
H(x,y,z) = x ^ y ^ z
I(x,y,z) = y ^ (x | ~z)
```

These operators are used in a loop kernel that iteratively applies the following four steps:

```
FF(a,b,c,d,M,s) => a = b + ((a + F(b,c,d) + M) << s)
GG(a,b,c,d,M,s) => a = b + ((a + G(b,c,d) + M) << s)
HH(a,b,c,d,M,s) => a = b + ((a + H(b,c,d) + M) << s)
II(a,b,c,d,M,s) => a = b + ((a + I(b,c,d) + M) << s)
```

We can think of applying customization at the different levels described in the previous sections.

In particular, we can add a fairly generic set of encryption-specific operations that we believe will also benefit other similar algorithms. Within the Lx architecture constraints we can easily add arithmetic operations that use more than two operands by bundling together multiple issue slots within a wide instruction word. In this case, a reasonable set of four-operand operations (of which we can issue two per cycle on a 1-cluster Lx) is the following:

```
ASM_1_F(a,x,y,z) = a + (x & y) | (~x & z)
ASM_1_G(a,x,y,z) = a + (x & z) | (y & ~z)
ASM_1_H(a,x,y,z) = a + x ^ y ^ z
ASM_1_I(a,x,y,z) = a + y ^ (x | ~z)
```

If MD5 is our one and only choice we can push customization one level further and actually implement the second set of more complex operations using six operands and three instruction slots (one per cycle, leaving the fourth slot free on a 1-cluster Lx):

```
ASM_2_FF(a,b,c,d,M,s) = b + ((a + F(b,c,d) + M) << s)
ASM_2_GG(a,b,c,d,M,s) = b + ((a + G(b,c,d) + M) << s)
ASM_2_HH(a,b,c,d,M,s) = b + ((a + H(b,c,d) + M) << s)
ASM_2_II(a,b,c,d,M,s) = b + ((a + I(b,c,d) + M) << s)
```

Table 3 shows the performance implications of these customizations (assuming a single-cycle implementation). In the left part of the table, we present performance numbers for MD5.

As we can see, customization is effective in this case: we gain a factor of 1.7x with the simple (level 1) customization, and a factor of 4.5x with the more complex (level 2) instructions.

The reason that customization can be dangerous is evident in the right half of Table 3, where we show the effect of the newly added instructions to another popular algorithm that implements an alternative one-way hashing technique, SHA (*Secure Hash Algorithm*). We can see how the level-1 (more generic) set of instruction still gives us a significant benefit (1.4x) for SHA, while the second (more specific) extension is not applicable to SHA. Note that since the level-2 set is a superset of level-1 we could still use it (with some effort) in the SHA case, achieving the performance represented by the numbers in parentheses. However, the level-2 extensions use a larger number of machine resources and while they are very effective for MD5, they are actually harmful for SHA performance.

In this example, if for any reason we were forced to switch to SHA, we would be much better off with the more generic (level-1) extensions. In practice, there are many reasons why this change may be unpredictable and occur late in the development process, for example if it turns out that the security of MD5 is not as we

	MD5				SHA			
	Cycles	Ops	ILP	Speedup vs. software	Cycles	Ops	ILP	Speedup vs. software
Software	445	640	1.44	**1.00**	384	1436	3.74	**1.00**
ASM level 1	262	350	1.34	**1.70**	259	784	3.03	**1.48**
ASM level 2	100	154	1.54	**4.45**	N/A (444)	N/A (784)	N/A (1.76)	**N/A (0.86)**

Table 3 Values are for a complete round of MD5 over 512 bits of input on a 1-cluster Lx.
The table includes data for the SHA algorithm (similar to MD5), using the same set of custom instructions designed for MD5. Values in parentheses use level-2 instructions to implement level-1 functionality.

expected when we started. To draw a parallel, a similar set of considerations is what caused MD4 (the MD5 predecessor) to be abandoned and replaced by MD5.

On the other hand, if MD5 is "the" standard, more aggressive customization brings major benefits. However, it is important to observe that for "standard" algorithms, the ASIC approach is indeed real competition to software-based technology. If the application is really fixed, it is certainly more cost-effective (and usually faster) to freeze the functionality in a piece of hardware. This further narrows the range of applicability for aggressive customization.

6. CONCLUSIONS

In this paper we presented the key features of the Lx technology platform. The lessons we learned from the design can be summarized as follows.

1. Domain-level specialization of embedded VLIW architectures is very effective, as the benchmarks that we presented indicate. The compiler technology is up to speed, and we have shown that cost/performance numbers are compelling. We have also demonstrated that the code size growth of VLIW architectures can be kept under control with careful design considerations.
2. Enabling scalability and customizability in a technology platform, requires many constraints in various aspects. For example, the Lx core ISA, run-time architecture (ABI), microarchitecture organization and pipeline have to be rigidly controlled. Lx is a good example of a design that takes these tradeoffs into account.
3. Scalability by increasing ILP resources is somewhat, but not uniformly, effective across all applications in a domain and requires some careful cost/performance analysis. The simplicity of a VLIW architecture makes scalability possible by increasing clock speed. This scales much more uniformly, although it impacts the power budget, which is usually limited in embedded domains. On pure general-purpose code we show that we can do as well as other competitive platforms at similar cost, but scalability does not apply (i.e. their ILP does not scale).
4. Aggressive customization works in limited cases, but it is dangerous to push because it is too application and algorithm-dependent, people want to be able to change the software at the last minute (for time-to-market), and it compromises concurrent hardware/software engineering. Despite automation, the cost of designing a high-performance VLIW core is still very high today and tools are not quite up to speed in a real-world production environment. Finally, issues like verification in the presence of user-driven customization are far from being solved and clearly point to research areas that are worth pursuing in the near future.

REFERENCES

[1] Colwell, R., O'Donnell, J., Papworth, D., and Rodman, P. "Instruction Storage Method with a Compressed Format using a Mask Word", U.S. Patent 5057837, Oct. 1991.

[2] Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages pages 180-192. ACM. 1987.

[3] Faraboschi, P., Fisher, J. and Desoli, G Clustered Instruction-Level Parallel Processors. *Hewlett-Packard Technical Report*. HPL-98-204, 1998.

[4] Fisher, J., Faraboschi, P., and Desoli, G. "Custom-Fit Processors: Letting Applications Define Architectures". In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO30)*, Paris, France, December 1996.

[5] Fisher, J. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Trans. on Computers*, C-30(7):478-490. 1981.

[6] IBM Corp. "CodePack Compression for PowerPC". Available as: http://www.chips.ibm.com/products/powerpc/cores/cdpak.html

[7] Lowney, P. G. et al. (1993). "The Multiflow Trace Scheduling Compiler". *The Journal of Supercomputing*, 7(1/2):51-142.

[8] Raik-Allen G. "ARC Cores rides platform divergence trend". Red Herring, June 1999. Available as http://www.redherring.com/insider/1999/0604/vcarccores.html

[9] Rau B., Yen D., Yen W., and Towle R., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs," IEEE Computer, January 1989, pp. 12-35.

[10] Schneier B. "Applied Cryptography (Second Edition). Protocols, Algorithms and Source Code in C". John Wiley and Sons. 1996.

[11] Sharangpani H. "Intel® Itanium Processor Microarchitecture Overview". Microprocessor Forum. 1999. Available as: http://developer.intel.com/design/ia-64/architecture.htm

[12] Slavenburg G, Rathnam S., Dijkstra H, "The TriMedia TM-1 PCI VLIW Media Processor", Hot Chips 8, August 1996.

[13] StarCore Alliance (Motorola Semiconductors and Lucent Technologies). Leadership in DSP Technology for Communications Applications. Available as: http://www.starcore-dsp.com/files/SC140pres.pdf

[14] Tensilica Inc., "Application Specific Microprocessor Solutions (Data Sheet for Xtensa V1)", 1998. Available as: http://www.tensilica.com/datasheet.pdf

[15] Texas Instruments Inc. "TMS320C6000: a High Performance DSP Platform". Available as: http://www.ti.com/sc/docs/products/dsp/c6000/index.htm

[16] Wolfe, A. and Chanin, A., "Executing Compressed Programs on An Embedded RISC Architecture", In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 81-91, Portland, Oregon